High Level Modeling of Elastic Circuits In SystemC

Mohamed Ammar Ben Khadra, Yu Bai and Klaus Schneider
Embedded Systems Group, Department of Computer Science
University of Kaiserslautern, Kaiserslautern, Germany
ammar@rhrk.uni-kl.de, {bai}|{schneider}@cs.uni-kl.de

Keywords: Elastic circuits, Synchronous languages, Quartz, SystemC, Model-based design

Abstract

Synchronous design is currently by far the mainstream design paradigm of digital circuits. However, the move to modern nano-meter technologies has brought unprecedented delay variability issues. That makes maintaining clock synchronization challenging and costly in terms of power and area. Elastic circuits is an emerging method for tackling delay variability while avoiding the technology disruption and design issues of asynchronous circuit design. We discuss a model-based approach to elastic circuits that starts from a system (circuit) specified in the synchronous language Quartz. The system is then elasticized i.e. partitioned to an elastic network consisting of inter-connected synchronous modules. The scope of this work is on synthesizing the elastic network by generating code for SysteMoC which is an actor-oriented modeling library based on SystemC. That enables rapid simulation of different partitioning and scheduling strategies at a high-level of abstraction. We have developed a synthesis library capable of generating code for most Quartz features. We show some experimental results based on synthesizing different synchronous models that have been elasticized using a basic partitioning strategy.

1. INTRODUCTION

Nowadays, the majority of digital systems design is based on the synchronous paradigm. Synchronous circuits rely on the availability of a control signal called a *clock*. The clock provides a global periodic reference to sequence circuit events. At an abstract level, a module of a synchronous circuit can be viewed as a set of state registers separated by a "cloud" of combinational logic as depicted in Figure 1. Note that the clock period must be greater than the highest signal propagation delay across **any** combinational logic cloud in the circuit. This condition sets an upper bound on the possible circuit frequency. Static Timing Analysis (STA) techniques are used to estimate the signal propagation delay in combinational logic.

Feature scaling to modern nano-meter technologies has brought numerous challenges to the synchronous paradigm including (1) manufacturing process variations where transistor characteristics vary widely between different dies or even across the same chip [1], (2) dominance of wire delay compared to gate delay which means that STA is not that effective anymore for worst-case delay estimation. Therefore, designers need to rely on costly iterations of synthesis and place & route to meet timing-closure, and (3) transistors are operated at a supply voltage that is close to their sub-threshold voltage in order to minimize power consumption. That means that transistors are more susceptible to run-time delay variation issues caused by voltage changes e. g. IR-drop.

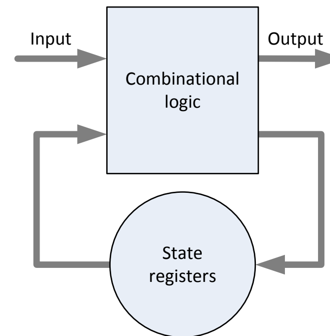


Figure 1: A model of a synchronous hardware module.

In an environment with such variations, distribution of a sharp clock signal with minimum skew and jitter is a significant challenge. Therefore, it is tempting to think in replacing the *global* clock based sequencing with *local* handshake based sequencing i. e. migrating to pure asynchronous design. That can potentially remove the high cost of clock tree distribution and provide resilience to delay variations. Unfortunately, the asynchronous design paradigm also suffers from another set of design issues since (1) it lakes CAD support and a mature general-purpose design flow [2], (2) it is disruptive to established technologies and practices especially since it requires designers to learn new, typically low-level, languages for circuit specification e. g. Balsa [3], and (3) it requires adding extra hardware for hazard suppression and completion detection. The required extra hardware can be quite large depending on the delay model.

The issues of asynchronous design motivate investigating other approaches that are more *evolutionary* i. e. adapting synchronous circuits to be elastic. In elastic circuits, computation and communication are considered separately which enables better tolerance to delay variations. Computation is preformed by *elastic modules* (EM) while communication be-

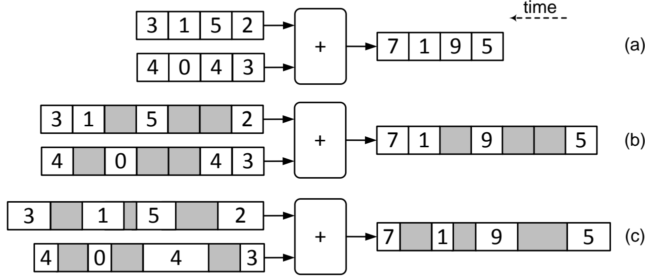


Figure 2: Comparing elasticity types were shaded fields refer to bubbles (a) synchronous circuit without elasticity, (b) synchronous elasticity, and (c) asynchronous elasticity.

tween elastic modules is done over *elastic channels* (EC) using a latency insensitive protocol instead of plane wires. Our considered EMs are synchronous i. e. their actions are driven by a clock signal. That enables taking advantage of the mature synchronous design flow in their design.

Basically, circuit elasticity can be classified to two different types depending on the communication style used in EC. Figure 2 compares different elasticity types to the original synchronous design. Basically, an EC is a limited capacity FIFO that consists of cells. A cell can hold a *token* (valid data) or a *bubble* (no data). Figure 2(a) depicts a traditional synchronous circuit where the duration of all cells is equal to the clock period and cells hold tokens at all times. That condition is relaxed in *synchronous elasticity* in Figure 2(b) where some cells can hold bubbles. That is relaxed even more in *asynchronous elasticity* of Figure 2(c) where cell duration can take arbitrary time instead of an exact clock-period. In this work, we focus on synchronous elasticity for communication between EMs since it's more efficient in hardware terms and doesn't require the overhead of sync/async domain interfacing of asynchronous elasticity. Nonetheless, we show how asynchronous elasticity can also be easily modeled.

Definition 1. ***Elastic circuit****: a delay-tolerant hardware circuit obtained by applying a FIFO-based correct-by-construction transformation to a synchronous circuit.*

Definition 1 emphasizes the fact that applying elasticity must not add any significant verification overhead to the synchronous design flow. That is, an elastic transformation must be correct-by-construction in order to be practical. Note that we haven't restricted the specification language of the synchronous system. Hence, it can be a synchronous language or traditional HDL, e. g. Verilog.

Additionally, note that Globally Asynchronous Local Synchronous Systems (GALS) [4], although they can tolerate delay variability, do not fall under definition 1 since they are not, typically, obtained using a correct-by-construction transformation to a given synchronous circuit. Moreover, EMs should be of a smaller granularity compared to GALS modules since their communication is based on simple FIFO-based protocols. Therefore, a two-level communication hierarchy can be imagined in future System-on-Chips. At the top level, modules utilize sophisticated buses and/or Network-on-Chips fabrics. In turn, circuit elasticity can be used for communication between sub-modules *within* those larger modules.

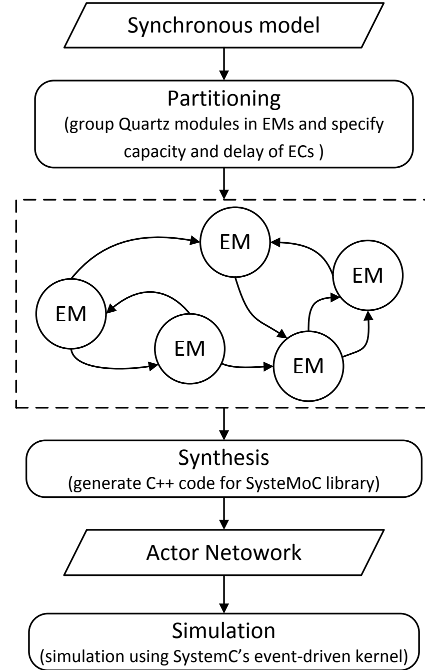


Figure 3: Flow of considered approach.

To fully establish our scope, it's imperative to note that elastic circuits can be produced using *de-synchornization*. It's a correct-by-construction transformation that is applied to the synthesized gate-netlist of a synchronous circuit to convert it to asynchronous one. For a good survey on de-synchronization approaches see [5]. However, de-synchronization is out of our scope since it moves us to the domain of pure asynchronous design which is what we are trying to avoid.

Definition 2. ***Patience property****: an elastic module is patient iff it doesn't fire until there are tokens on all of its input ports and bubbles on all of its output ports.*

We model an EC as FIFO from a source port (output of an EM) to a destination port (input of another EM). The FIFO has minimum propagation delay and finite capacity. At each clock cycle, a token can advance iff the next cell holds a bubble i. e. tokens can't override each other. That creates back-pressure on the producing EM to prevent it from firing if the FIFO is full. The maximum time (number of clock cycles) a token spends in an EC depends on the firing of the con-

suming EM. Definition 2 is of a special importance to us in the context of EC modeling. It has been proved by Carloni et al. [6, 7] that EMs satisfying the patience property are not affected by varying delay (and as consequence also capacity) of the EC. Therefore, we can consider computation and communication in isolation. That makes elastic circuits well-suited to be modeled as actor networks [8], which is the high level model considered in our approach.

The flow of our model-based approach is depicted in Figure 3. It starts by specifying the system in the synchronous language Quartz [9] which is the modeling language of our Averest[1] framework. Then, the system is partitioned to an elastic network of EMs connected by ECs. Formally,

Definition 3. ***Elastic module****: is the tuple* $(\mathcal{P}, \mathcal{L}, \mathcal{CF}, \mathcal{DF})$ *where* $\mathcal{P}$ *is a set of input and output ports* $I \cup O$*.* $\mathcal{L}$ *is the set of local variables which define the state of the module.* $\mathcal{CF}$ *and* $\mathcal{DF}$ *define sets of control-flow and data-flow guarded actions respectively.*

Definition 4. ***Elastic channel****: is the tuple* $(sP, dP, \mathcal{D}, \mathcal{C})$ *where sP is a source port of an EM and dP is a destination port.* $\mathcal{D}$ *is the minimum token delay.* $\mathcal{C}$ *is the maximum capacity (number of tokens) of the channel.*

The behavior of EM's is described by Guarded Actions (GA) which are designed in the spirit of guarded commands [10], a well-established formalism for specifying concurrent systems. We separate an EM's guarded actions to Control-flow Guarded Actions (CGA) and Data-flow Guarded Actions (DGA). Compilation and partitioning of Quartz models is beyond our scope. We are primarily concerned with synthesizing our abstract elastic network model to an actor network. Then, the generated actor network can be simulated using SystemC's discrete-event simulation kernel.

SystemC [11] is an open source C++ library used for Electronic System Level (ESL) design of embedded system. It extends C++ with hardware related classes and data types. Nowadays, it's the de-facto standard for Hardware/Software co-design. Instead of generating SystemC code directly, we used SysteMoC [12] which is an open-source actor-oriented modeling library based on SystemC. It enabled us to natively express elastic circuits as actor networks. It also enable us to utilize the analysis that has already been developed for it [13].

The main contribution of this work is in the modeling of elastic circuits as networks consisting of EMs that communicate over ECs. We show how that abstract model can be synthesized to SysteMoC. We also discuss how this model can be mapped to actual hardware. To our knowledge, modeling and simulation of elastic circuits in SystemC has not been considered before. Our discussion proceeds as follows, we discuss some related work in section 2. Then, we give in section 3. the necessary background on synchronous modeling and actor-oriented modeling. Later, we show in section 4. how SysteMoC simulation code for an EM and an EC can be generated. We also show how an EM and an EC can be mapped to actual hardware. Finally, we conclude with some experimental results based on a simple partitioning scheme for several synchronous model benchmarks.

[1] Available at http://www.averest.org/

2. RELATED WORK

There are two main approaches to elastic circuits proposed in the literature, namely, *elastic circuits* proposed by Cortadella et al. [14–16] and *Latency Insensitive Protocols* (LIP) proposed by Carloni et al. [6, 7]. In the former approach, elasticity has been addressed at a fine-grained Register-Transfer Level (RTL). RTL transformations have been proposed in [15] to a synthesized gate-netlist. These transformations should be considered in addition to well-known RTL transformations like retiming. This approach requires considerable support from the synthesis as well as place & route tools. However, it is of special importance to us since it's where the Synchronous ELastic Flow (SELF) protocol has been proposed [14]. SELF is our target protocol for the hardware synthesis of ECs.

In LIP, it has been proposed to wrap synthesized synchronous modules (pearls) with communication wrappers (shells) that enables them to communicate using ports over a latency-insensitive channel. It proposes a protocol similar to SELF but less specified. Their protocol employs so-called *relay stations*. LIP is less disruptive than the former approach in the sense that it requires less support from the synthesis tool. Note that both approaches were not primarily concerned with the modeling of elastic circuits for simulation purposes. Actually, Petri-nets is the main simulation tool for the former approach as demonstrated in [17]. It's used for gate-netlist modeling which is obviously a very low level. In our approach, EMs can model hardware modules seamlessly at various granularity levels from simple combinational gates to Intellectual Property (IP) cores or even complete processors.

Brandt et al. have considered in [18] the synthesis of Quartz synchronous models to SystemC. Compared to their work, we show how an elasticized (partitioned) synchronous model can be synthesized and simulated in SysteMoC in the context of elastic circuits. We not only bridge the semantics gap but also show a flexible way for simulating the SELF protocol at high level of abstraction. Additionally, Hoover et al. have discussed in [19] the synthesis of elastic data-flow (actor) networks from atomic guarded actions. Their proposed language is a mix of guarded actions and Verilog code. Verilog was used to describe combinational logic. Our guarded actions representation is compiled from Quartz, a modeling language with precise formal semantics that enables formal verification in addition to synthesis to hardware as well as to software.

```
module MAC(int ?a, ?b, !s) {
 int i, t, o;
 loop {
   w1: pause;
   t = i;
   o = a*t;
 }
 ||
 loop {
   w2: pause;
   i = a+b;
   if (t < 0) {
     w3: pause;
     s = o;
   } else {
     w4: pause;
     if (b>0)
       s = o + 1;
}}}
```

$$\begin{bmatrix}
\alpha_1: & start \vee w_1 & \Rightarrow & next(w1) = \mathsf{true} \\
\alpha_2: & start \vee w_3 \vee w_4 & \Rightarrow & next(w2) = \mathsf{true} \\
\alpha_3: & w_2 \wedge (t < 0) & \Rightarrow & next(w3) = \mathsf{true} \\
\alpha_4: & w_2 \wedge \neg(t < 0) & \Rightarrow & next(w4) = \mathsf{true} \\
 & & & \\
\beta_1: & w_1 & \Rightarrow & t = i \\
\beta_2: & w_1 & \Rightarrow & o = a \times t \\
\beta_3: & w_2 & \Rightarrow & i = a + b \\
\beta_4: & w_3 & \Rightarrow & s = o \\
\beta_5: & w_4 \wedge (b > 0) & \Rightarrow & s = o + 1
\end{bmatrix}$$

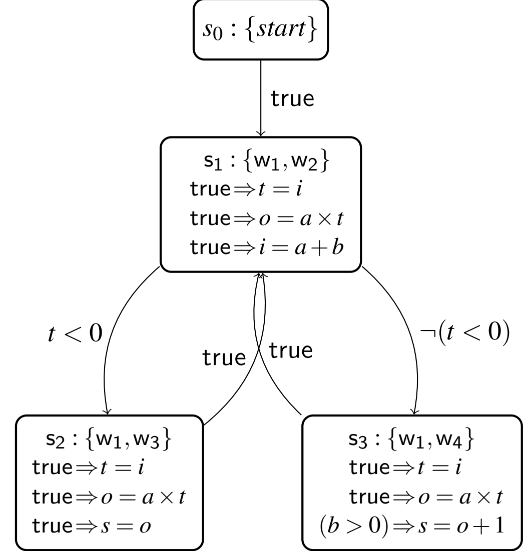


Figure 4: (left) Quartz module MAC, (middle) its guarded actions consisting of CGAs(α_i) and DGAs (β_i), and (right) generated EFSM.

3. BACKGROUND

We discuss Quartz model transformation. Then, we show how the transformed model can be represented in SysteMoC.

3.1. Synchronous modeling in Quartz

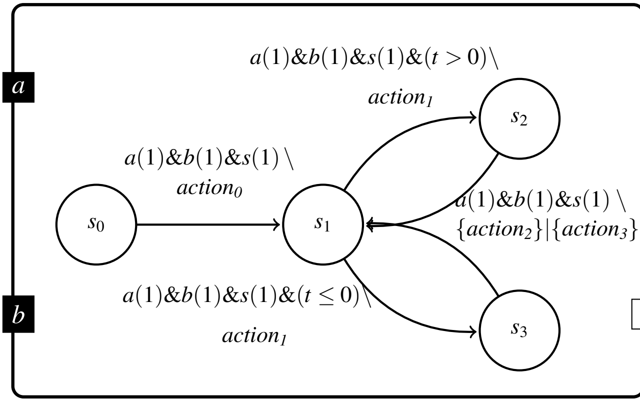
Synchronous languages [20] have emerged since early eighties to address the specification issues of reactive systems. They are based on a simple Model of Computation (MoC) hypothesis of *perfect synchrony* where the execution of the system is divided into discrete reaction steps called *macro-steps*. In each macro-step, the system reads all inputs, executes a finite number of *micro-steps*, and finally produces outputs w.r.t. internal system state. All micro-steps are executed in the same variable environment i. e. a variable can take only one value in a macro-step. Micro-steps are assumed to consume no time, and time advances to the next macro-step after all micro-steps are finished. Simplicity of the synchronous hypothesis enables easier reasoning about system behavior. Additionally, it lends itself naturally to formal verification e. g. using model checking.

Figure 4 depicts our example Quartz module MAC with input variables `a` and `b` and output variable `s` and local variables `i`, `t`, `o`. Macro-steps are determined using `pause` statements which have been labeled to show the control-flow of the module. Basically, MAC consists of two infinite loops running in parallel. However, they are running in lock-steps synchronizing at each `pause`.

Compiling MAC using Averest's Quartz compiler yields a set of CGAs ($\mathcal{CF}$) and a set of DGAs ($\mathcal{DF}$) as depicted in Figure 4(middle). A guarded action has the form $\langle \gamma \Rightarrow \alpha \rangle$, where γ is a boolean expression *guard* and α an *assignment*. An action is executed only when its corresponding guard is satisfied. An *immediate assignment* action, denoted by $\langle x = e \rangle$, assigns x to the evaluated result of expression e in the current macro-step, while a *delayed assignment* denoted by $\langle next(x) = e \rangle$, assigns x to the value of e in current macro-step. However, the assignment is executed in the next macro-step. Note that most synchronous languages can be compiled to guarded actions which makes our characterization of EM's behavior in terms of $\mathcal{CF}$ and $\mathcal{DF}$ a common intermediate format that is not limited to Quartz.

One can represent a synchronous system with a state-machine that has a single state with all DGAs attached to it. At each clock tick, all guards γ are evaluated. Then, only the assignments α that have their γ evaluated to true will be executed. However, we are interested in a more efficient representation (in terms of computation) of the system by only evaluating DGAs that belong the current reaction. To this end, we generate an Extended Finite Machine from the given CGAs of the system. Then, each DGA is attached only to the state(s) where it could possibly be executed. EFSM is defined formally in Definition 5.

Definition 5. ***Extended Finite State Machine (EFSM)****: is the tuple $(S, s0, T, D)$, where S is a set of states, $s0 \in S$ is the initial state, and $T \subseteq (S \times G \times S)$ is a finite set of transition relations where G is the set of transition guards. D is a mapping $S \rightarrow D$, which assigns each state $s \in S$ a set of DGAs $D(s) \subseteq D$ which are executed in state s.*




```
class MAC : public smoc_actor {
    SC_HAS_PROCESS(MAC);
  smoc_firing_state s0, s1, s2, s3;
  // Other definitions omitted.
public:
  smoc_port_in<int> a, b; smoc_port_out<int> s;
  MAC(sc_module_name name)
    : // Constructor instantiations omitted.
  {
  // Only firing transitions of s1 are shown.
    s1=TILL(clk)>>(a(1)&&b(1)&&(guard1))>>s(1)
        >>CALL(MAC::action1)>>s2|
        TILL(clk)>>(a(1)&&b(1)&&(guard2))>>s(1)
        >>CALL(MAC::action1)>>s3;
 } };
```

Figure 5: SysteMoC actor model of module MAC, (left) previously generated EFSM now annotated with SysteMoC firing guards. Each $action_i$ executes DGAs corresponding to EFSM state s_i, patience property is observed by guarding all firing transitions such that one token (bubble) should be available on all input (output) ports before firing, (right) textual C++ code generated for the actor where **guard1** and **guard2** are functions $(t>0)$ and $(t\leq 0)$ respectively.

Figure 4 depicts the generated EFSM of module MAC. The default starting state is s_0 where only label *start* is set. Each state has been annotated with its control flow labels and its attached DGAs.

3.2. Actor-oriented modeling in SysteMoC

SysteMoC is used to describe a network graph of communicating actors. Each actor has a set of input ports I and a set output ports O. Ports have a supported token type e. g. double. An actor's input port should be connected to the output port of another actor that support the same token type. The connection is a FIFO that has a configurable size. Actor's internal state is defined by a set of local variables $\mathcal{L}'$ that are readable and writable only inside the actor. The behavior of the actor is defined by a Firing-FSM (FFSM). Formally,

Definition 6. ***Actor network graph****: is a directed bipartite graph (A,C,P,E) containing a set of actors A, a set of channels C, a channel parameter function $P: C \rightarrow N^{\infty} \times V^*$ which associates with each channel $c \in C$ its buffer size $n \in N^{\infty} = \{1,2,3,...\infty\}$, and possibly also a non-empty sequence $v \in V$ of initial tokens , and finally a set of directed edges $E \subseteq (C \times A.I) \cup (A.O \times C)$. The edges are further constraint such that exactly one edge is incident to each actor port and the in-degree and out-degree of each channel in the graph is exactly one.*

Definition 7. ***Actor****: is the tuple $(\mathcal{P},\mathcal{L}',\mathcal{F},\mathcal{R})$ where $\mathcal{P}$ is a set of input and output ports $\mathcal{P} = I \cup O$, $\mathcal{L}'$ is the set of local variables which define the state of the actor. $\mathcal{F}$ is a set of functions, $\mathcal{R}$ is the firing FSM.*

Note that $\mathcal{F} = \mathcal{F}_G \cup \mathcal{F}_A$ where $\mathcal{F}_G$ is a set boolean functions used to guard firing transitions, and $\mathcal{F}_A$ is set of firing actions executed upon firing. Only $\mathcal{F}_A$ are able to change values of $\mathcal{L}'$. Note also that an EFSM is a restricted form of a FFSM where the relation between FSM states and $\mathcal{F}_A$ is bijective. Therefore, we describe the behavior of actors using EFSM. Guards of firing transitions are described by $G \subseteq \mathcal{G}_{clk} \times \mathcal{G}_I \times \mathcal{G}_O \times \mathcal{F}_G$, where $\mathcal{G}_I$ ($\mathcal{G}_O$) are used to guard that sufficient number of tokens (bubbles) are available on input (output) ports to be consumed (produced), and $\mathcal{G}_{clk}$ is a clock event condition used if clock synchronization is required. Clocks in SysteMoC are defined per actor. Therefore, firing transitions can be synchronized by setting an equal clock period (main clock) for all actors in the network and setting all firing transitions to have $\mathcal{G}_{clk}$. It is simple to model actors that require multi-cycles for their computation by setting their clock to an integer number of the main clock period. It is even possible to configure delay time for individual firing transitions by using Virtual Processing Component (VPC) which is a supporting framework to SysteMoC.

4. SYNTHESIS OF MODEL ENTITIES

We discuss here modeling and code-generation of an Elastic Module and later of an Elastic Channel.

4.1. Synthesis of an Elastic Module

Given an EM defined by $(\mathcal{P},\mathcal{L},\mathcal{CF},\mathcal{DF})$ we need to generate its corresponding SysteMoC actor $(\mathcal{P},\mathcal{L}',\mathcal{F},\mathcal{R})$. To this end, we need to generate and synthesize the EFSM $\mathcal{R}$ based on the given $(\mathcal{CF},\mathcal{DF})$. Additionally, we synthesize $\mathcal{F} = \mathcal{F}_G \cup \mathcal{F}_A$ where each firing action in $\mathcal{F}_A$ is generated from DGAs of a corresponding state in $\mathcal{R}$, while $\mathcal{F}_G$ are generated from transition guards of $\mathcal{R}$. Finally, actor variables have to be synthesized such that $\mathcal{L}' = \mathcal{L} \cup \mathcal{L}_E$ where $\mathcal{L}_E$ are extra vari-

ables required to handle the semantic mismatch between the two different MoCs e. g. output variable are writable only in actors whereas they are both readable and writable in Quartz. The details of this synthesis procedure have been omitted from this work due to the lack of space. We refer the interested reader to [21] for a more detailed treatment.

Figure 5 shows a graphical and a textual representation of the actor representing module MAC. Note that $\mathcal{G}_I$ and $\mathcal{G}_O$ of all firing transitions must insure that there is one token (bubble) on all input (output) ports before firing. That is required to observe the patience property. Additionally, all firing transitions should be guarded with a clock condition $\mathcal{G}_{clk}$ to synchronize firing of all actors in the network.

Figure 6 depicts the target hardware model of our considered EM. Compared to the model of traditional synchronous modules of Figure 1, one can note that combinational logic has been replaced by Trigger Logic (TL) and Computation Logic (CL). At its simplest forms, TL would implement SELF protocol communication with other EMs and control the patience property, while CL would implement the combinational logic of the original synchronous module. Maintaining patience requires clock-gating the registers to keep their state which saves dynamic power. Additionally, note that output of CL may be connected to a traditional synchronous module instead of an EM.

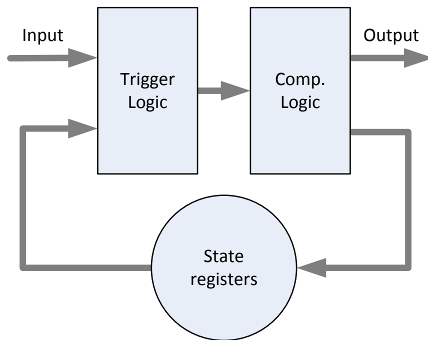


Figure 6: A hardware model of an elastic hardware module

We believe that synthesis of EMs should be considered early in design since many potential functions of TL may require considering TL and CL together for synthesis. Therefore, we argue against black-box wrapping approaches that considers circuit elasticity only later, e. g. LIP [6]. Basically, functions implemented by TL can go well beyond maintaining patience to include (1) multi-cycle control where TL can analyze a given input and determine if it requires one or more cycles to compute in CL and keep patience-mode activated accordingly (2) power-gating of CL in case no computations are required which saves static power. Actually, saving static power at this fine-grained (per-computation) level is particularly important technique for handling the utilization wall

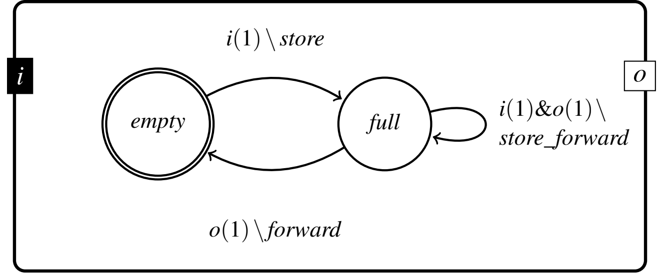


Figure 7: Actor model of a Channel Buffer

[22]. Basically, the utilization wall observation states that the percentage of chip area that can be powered-on within chip's power budget is dropping exponentially with the progression of Moore's law and thus the rest of the chip forms so-called *dark silicon*. Based on that, one should consider partitioning of CL to multiple-parts, even at the cost of more redundancy, where each part is responsible of executing one (or more) firing actions. Therefore, TL would *route* input to the required CL part while keeping other parts power-gated. We believe that it's very difficult to consider CL partitioning and input routing in circuits specified in HDL due to the relatively low abstraction level and the lack of formal semantics. Hence, it is reasonable to consider model-based design in that context.

4.2. Synthesis of an Elastic Channel

We consider here the synthesis of an EC defined by the tuple $(sP, dP, \mathcal{D}, \mathcal{C})$. Note that it is important for us to make EMs completely independent of ECs. Hence, we can use the same EM with different channel configuration of $\mathcal{D}$ (delay) and $\mathcal{C}$ (capacity). Unfortunately, SysteMoC only supports asynchronous communication using a FIFO class named **`smoc_fifo`**. Consequently, a **`smoc_fifo`** can be used to to model an EC with $\mathcal{D} = 0$. In order to simulate configurable clock cycle delay on an EC we created a special actor named Channel Buffer (CB). Basically, a CB is a SysteMoC actor that has the sole purpose of delaying its input by one clock cycle. Therefore, to model an EC delay of $\mathcal{D}$ clock cycles, one needs to chain $\mathcal{D}$ number of CBs together. The actor model of a CB is depicted in Figure 7.

Token storage is implemented by a **`smoc_fifo`** connecting two CBs. Note that $\mathcal{D}$ and $\mathcal{C}$ of an EC can't be considered independently. We need to keep hardware synthesizability to our target SELF protocol in mind. To this end, we have to make sure that (1) the configured maximum capacity of a **`smoc_fifo`** connecting two CBs is either one or two tokens, and (2) that $\mathcal{D} < \mathcal{C} \leq 2\mathcal{D}$. These conditions stem from the fact that SELF is based on chaining Elastic Buffers (EB) and/or Elastic Half Buffers (EHB) both with a delay of one clock cycle. However, EB have a storage capacity of two tokens compared to one token in EHB. For more details on SELF

and its hardware implementation options see [14, 16]. Figure 8 compares two ECs with the same $\mathcal{D}$ and different $\mathcal{C}$. Note that we generated a single C++ template class for CB. The compiler would then instantiate as many classes as needed depending on the declared token types e.g. boolean type.

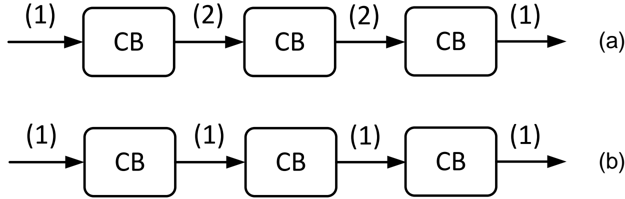


Figure 8: Two ECs with the same $\mathcal{D} = 3$ and different $\mathcal{C}$, (a) set with maximum $\mathcal{C} = 6$ which is mapped to a chain of EBs in SELF and (b) set with minimum $\mathcal{C} = 4$ which is mapped to an EB connected to a chain of EHBs in SELF.

Finally, join and fork nodes of the SELF protocol can be implemented by corresponding actors. Additionally, there is also the possibility to model $\mathcal{D}$ and $\mathcal{C}$ of the EC using a single complex CB instead of our chain of simple CBs. However, the C++ compiler will then need to instantiate a different class for each different combination of $\mathcal{D}$, $\mathcal{C}$ and token type. Note also that asynchronous elasticity can be modeled by using a single CB that has clock period set to arbitrary time rather than set to the main clock period as in synchronous elasticity.

5. EXPERIMENTAL RESULTS

We have developed a synthesis library that is capable of generating code for most features of Quartz. The library was developed in F#.NET and it has about 3900 Lines of Code (LoC). We took advantage of the data-types already supported by SystemC e. g. bitvictors (**`sc_bv`**). Synthesis of aggregate data-types (tuples) was also straightforward e. g. tuples where mapped to C++ structs. We also generated **`assert`** statements to make sure that the original specifications won't be violated at run-time e. g. out-of-bound array access.

We consider here a basic partitioning strategy where the synchronous system is partitioned to two modules, namely, one that provides input stimuli (driver) and another that reacts based on that input (main). Therefore, the resulting actor network consists of two actors representing two EMs. Example results of the experiments conducted on different benchmarks is given in Table 1. We list the example alongside, the time required to generate code for it, the effective number of LoC, the number of canonicalized boolean expressions during EFSM generation, and the total number of states in the EFSM. We listed the number of canonicalized boolean expressions since it's the most expensive computational operation. Basically, expression canonicalization is used to generate deterministic firing transitions among other things.

The tested benchmarks were Heron (Newton) square root algorithm, a simple car cruise control model, and an implementation of SHA2-256 hashing algorithm. Thanks to the robust support of bitvictors in SystemC, we haven't had any issue in synthesizing the SHA2 model, although it utilizes some sophisticated bitvictor operations. SHA2 model was implemented in two versions, a basic version that maps the standard directly and uses all 64 scheduling values W_t, and an optimized version that starts hashing a block as soon as a hash word has been received. It uses the last 16 scheduling values only which is typical for commercial cores. The optimized version has more states since it requires more control. The generated code is then compiled using `g++` and linked against SysteMoC, SystemC and some Boost libraries. Comparing SHA's basic and optimized versions, it's clear that LoC number grows rapidly. That is due to having more EFSM states, consequently more firing actions in generated code, and more guarded actions attached to each state. Sharing of guarded action code between different firing action can be employed to reduce code size.

Table 1: Experimentation results

Model	Time	LoC	Boolexps	States
Heron Sqr Root	0.1 s	462	11	3
Cruise Control	1 s	1266	335	11
SHA (Basic)	3 s	5076	553	60
SHA (Optimized)	12 s	38012	7301	163

6. CONCLUSION

We have shown how elastic circuits can be modeled in SysteMoC using actor-oriented modeling. That allows taking advantage of the powerful modeling features of SystemC including its robust open source simulation kernel. Moreover, simulation of synchronous modules is supported at various granularity levels. We have also discussed the significant potential of elastic circuit design in handling the utilization wall. Based on that, our future work will go in three main directions. Firstly, we need to address the issue of optimized partitioning of synchronous models w.r.t. a given criteria. Secondly, we need to consider the customization of the EC based on the actual computations required by EMs e. g. if computations can be schedulable, we can remove the back-pressure channel required by SELF protocol and thus save hardware resources. Thirdly, our discussion of elastic circuits has been based on the patience property. However, we need to consider early evaluation where an EM fires as soon as *sufficient* input tokens are available instead of waiting for *all* inputs. In that way, we can bring elastic circuits closer to the average-case performance of asynchronous circuits while avoiding their disadvantages.

ACKNOWLEDGMENT

We would like to thank Joachim Falk of the HW/SW Co-Design group at Erlangen-Nuremberg University for providing us with SysteMoC 1.0 beta version. This version supports clock synchronization and it is still not publicly released to the time of this writing.

REFERENCES

[1] K. Bowman, S. Duvall, and J. Meindl, "Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration," *IEEE Journal of Solid-State Circuits*, vol. 37, no. 2, pp. 183–190, 2002.

[2] J. Sparso, "Current trends in high-level synthesis of asynchronous circuits," in *2009 16th IEEE International Conference on Electronics, Circuits and Systems - (ICECS 2009)*, pp. 347–350, IEEE, Dec. 2009.

[3] D. Edwards and A. Bardsley, "Balsa: An Asynchronous Hardware Synthesis Language," *The Computer Journal*, vol. 45, pp. 12–18, Jan. 2002.

[4] D. M. Chapiro, *Globally-Asynchronous Locally-Synchronous Systems*. Phd thesis, Standford University, Oct. 1984.

[5] M. Simlastik and V. Stopjakova, "Automated Synchronous-to-Asynchronous Circuits Conversion: A Survey," in *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation* (L. Svensson and J. Monteiro, eds.), vol. 5349 of *Lecture Notes in Computer Science*, pp. 348–358, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[6] L. P. Carloni and K. L. Mcmillan, "Latency Insensitive Protocols," in *Computer Aided Verification* (N. Halbwachs and D. Peled, eds.), vol. 1633, pp. 123–133, Springer Berlin Heidelberg, 1999.

[7] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1059–1076, 2001.

[8] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott, "A foundation for actor computation," *Journal of Functional Programming*, vol. 7, pp. 1–72, Jan. 1997.

[9] K. Schneider, "The synchronous programming language Quartz," Tech. Rep. 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2009.

[10] K. M. Chandy and J. Misra, *Parallel program design: a foundation*. Austin, Texas, USA: Addison-Wesley, 1988.

[11] Committee, *IEEE Standard 1666-2011 for SystemC Language Reference Manual*. New Jersey, USA: IEEE Standards Association, 2012.

[12] J. Falk, C. Haubelt, and J. Teich, "Efficient Representation and Simulation of Model-Based Designs in SystemC," in *Proceedings of Forum on Specification and Design Languages 2006*, FDL 2006, (Darmstadt, Germany), pp. 129–134, ECSI, Sept. 2006.

[13] J. Falk, C. Zebelein, J. Keinert, C. Haubelt, J. Teich, and S. S. Bhattacharyya, "Analysis of SystemC actor networks for efficient synthesis," *ACM Transactions on Embedded Computing Systems*, vol. 10, pp. 1–34, Dec. 2010.

[14] J. Cortadella, M. Kishinevsky, and B. Grundmann, "Self: Specification and design of synchronous elastic circuits," in *TAU 06: Proceedings of the ACM/IEEE International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, 2006.

[15] J. Cortadella, M. Galceran-Oms, and M. Kishinevsky, "Elastic systems," in *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, pp. 149–158, IEEE, July 2010.

[16] J. Carmona, J. Cortadella, M. Kishinevsky, and A. Taubin, "Elastic Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, pp. 1437–1455, Oct. 2009.

[17] J. Cortadella, M. Kishinevsky, D. Bufistov, J. Carmona, and J. Júlvez, "Elasticity and Petri Nets," in *Transactions on Petri Nets and Other Models of Concurrency I* (K. Jensen, W. M. P. Aalst, and J. Billington, eds.), vol. 5100 of *Lecture Notes in Computer Science*, pp. 221–249, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[18] J. Brandt, M. Gemuende, and K. Schneider, "From Synchronous Guarded Actions to SystemC," in *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)* (M. Dietrich, ed.), (Dresden, Germany), pp. 187–196, Fraunhofer Verlag, 2010.

[19] G. Hoover and F. Brewer, "Synthesizing Synchronous Elastic Flow Networks," in *2008 Design, Automation and Test in Europe*, pp. 306–311, IEEE, Mar. 2008.

[20] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, pp. 64–83, Jan. 2003.

[21] M. A. Ben Khadra, *A Model-Based Approach to Synchronous Elastic Systems*. Master's thesis, University of Kaiserslautern, Oct. 2013.

[22] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, (New York, NY, USA), pp. 365–376, ACM, 2011.